SOFTWARE DEVELOPMENT TOOLS

Glenn P. Forney
Walter W. Jones

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
National Engineering Laboratory
Center for Fire Research
Gaithersburg, MD 20899

NIST

SOFTWARE DEVELOPMENT TOOLS

Glenn P. Forney
Walter W. Jones

.

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
National Engineering Laboratory
Center for Fire Research
Gaithersburg, MD 20899

CONTENTS

Software Development Tools

Glenn P. Forney
Walter W. Jones

Center for Fire Research
National Institute of Standards and Technology

This paper discusses the use of software tools to aid in the development of models produced by the Center for Fire Research (CFR). There are two types of tools described in this paper. The first type consists of executable programs that characterize the subroutine and data structures of FORTRAN programs. A second class of software tools are subroutines that support various utility functions required by CFR models. We will discuss how each of these tools are used and why their development was necessary. In addition, we will indicate how these tools might be improved.

1. INTRODUCTION

This paper discusses how software tools are used to help develop fire growth and smoke spread models produced at the Center for Fire Research (CFR). There are two types of tools described in this paper. The first type consists of stand-alone or executable programs that characterize the subroutine and data structures of FORTRAN programs. A second class of software tools are subroutines that support various utility functions required by CFR fire models. We will discuss how each of these tools are used and why their development was necessary.

The underlying theme is the reuse of reliable routines. In general, programs use only a few techniques to solve most of the problems encountered in programming. Once a reliable routine has been developed, much time can be saved by reusing this routine rather than rewriting it, with the attendant debugging and documentation. This is an attempt to document a few such routines which the authors have found to be generally useful.

The stand-alone tools, ROADMAP, FDIFF and COMCHECK examine a program's procedure and data structures. They help to identify and eliminate problems that may exist in the subroutines and data structures. ROADMAP determines how subroutines are related to each other by reporting the ancestors and descendants of each subroutine in a program.

FDIFF compares two FORTRAN programs. Its primary usage is to document differences in versions as a module develops over time. COMCHECK determines how common blocks variables are used (or misused) in a program.

A second category of software tools are subroutines that support utility functions required by CFR fire models. Some of these subroutines provide an interface between a FORTRAN program and the graphical capabilities of computers and terminals. Other subroutines parse text, examine character strings, manage memory and access files. These tasks are required by the tools described above and fire models such as FAST[1], CFAST[2] and CCFM.VENTS[3].

These two types of tools are designed to increase the productivity of a software developer by shortening the time required to understand how a program is structured, identifying differences between similar versions of a program, determining where and how common block variables are used and providing software building blocks to be used to write other programs. These tools are also designed to make it easier to move procedures or algorithms from one project to another, and to insure that their interfaces are consistent. Both types of tools have been used on a wide variety of computers, thus attesting to their portability. They have been used on several development projects, so they can claim re-usability.

2. PROGRAM DEVELOPMENT TOOLS

The program development tools, ROADMAP, COMCHECK and FDIFF provide information about a program's routine and data structures. The tools ROADMAP and COMCHECK were designed to read a cross-reference[1] listing produced by a FORTRAN compiler. ROADMAP was originally written to document the structure of CCFM.VENTS[3] and was designed to examine a Cyber 855 FORTRAN listing. It was adapted to examine FORTRAN programs on an MSDOS compatible micro-computer so that it could be used to examine the structure of CFAST[2]. COMCHECK was written to determine where common block variables are being changed.

FDIFF was written to compare two FORTRAN programs. What sets it apart from other text comparison programs is that it uses the fact that it is examining FORTRAN programs. It can optionally ignore blanks and comment statements since changes in this area will not affect the results of a FORTRAN compilation. It also examines each file to be compared subroutine by subroutine. This tool was used to indicate which subroutines in the input, model and plot portions of CFAST were essentially identical.

Each tool can be executed from the command line in MSDOS. The software tools read in one or more files and produce an output file. Since similar structures exist for other programming languages, and operating systems, FDIFF, COMCHECK and ROADMAP could easily be ported to C, Pascal or Ada.

2.1 Analyzing Program Structures, ROADMAP

When working with small programs it is easy for the original developer to keep the plan or roadmap of the program in his head. As a program grows in size and complexity it becomes increasingly difficult to understand how it is structured by just examining a source listing. This is especially true for newcomers to a programming project. Further, when making changes to the subroutine or data structures in a computer program it is necessary to know the effect these changes will have on the rest of the program. The tool, ROADMAP, provides an automatic means of determining what other parts of a program need to be altered. Obtaining this information by hand always leaves the possibility that a required change will be missed. An example will help illustrate the point. Suppose that a subroutine XYZ has three arguments as in:

---

[1]   A cross-reference listing indicates how and where each subroutine symbol (variable, statement label, subroutine name, common block name) is used.

3

SUBROUTINE XYZ(A,B,C).

The developer wishes to add a variable to the argument list so that the new subroutine statement is given by

SUBROUTINE XYZ(A,B,C,D).

To complete the change, every location in the program that "calls" XYZ must be located so that statements of the form

CALL XYZ(A,B,C)

can be changed to

CALL XYZ(A,B,C,D).

This information, the routines that call XYZ, is not readily available from a program's source listing. A text editor, a program for editing text files, could be used to search for occurrences of the string "CALL XYZ" This is a tedious and error prone process. A simple text search would not find the string "CALL   XYZ" since more than one blank exists between CALL and XYZ. The tool, ROADMAP, provides an automatic means of providing the required information by listing each routine that calls a given routine. The usage of ROADMAP is given below. Parameters in brackets are optional.

ROADMAP   The program, ROADMAP, displays information about subroutine usage in a FORTRAN program. It determines the structure of a program by printing for each subroutine the routines that call it, the routines that it calls and the common blocks that are referenced. ROADMAP also determines the routines that access common blocks. The file names file1 and file2 may be any valid MSDOS file name. Further, file1 may contain MSDOS wild-card characters, '*' and/or '?' to specify a list of filenames to be examined. The characters '..', '.' may be used to represent the parent and current directory respectively in a path used to specify a file.

syntax:

ROADMAP file1 [file2]

file1   *required* File containing a cross-reference listing. This file must be generated by compiling a FORTRAN program using the Lahey FORTRAN compiler with the XREF listing option turned on.

file2   *optional* File containing a report produced by this program.The default name for this file is the same name as file$_1$ with the extension .RDM.

To examine the structure of a program using ROADMAP, consider program ONE given in the appendix. Suppose it is located in the file ONE.FOR. At an MSDOS prompt,

type:

F77L   ONE

making sure that the X-REF options has been turned on. F77L is the name of Lahey's FORTRAN compiler. A file named ONE.LST will be created. Next type:

ROADMAP   ONE.LST

This command creates the file ONE.RDM which contains ROADMAP's analysis of the program ONE.

File names used with ROADMAP may in general be full path names containing wild card characters such as '*' or '?' and symbolic directory names such as the parent directory, '..', or the current directory, '.'. To generate a ROADMAP for all subroutines in a directory named \PROG, first compile each routine in \PROG. Next type

ROADMAP \PROG*.lst CPROG.rdm

ROADMAP will input each cross-reference listing found in the directory \PROG and output the result to the file CPROG.RDM .

Each subroutine entry, NAME, in the ROADMAP output has up to four sub-headings: CALLS, LIB, COMMONS and CALLED BY. The subheadings CALLS and LIB are similar. They both list external references to NAME., *i.e.*, routines that NAME calls. The source code for routines that are listed under CALLS appeared in cross-reference listings that ROADMAP examined. On the other hand, the source code for routines listed by the LIB sub-heading did not. Some examples of routines that would appear under LIB are functions that are specific to FORTRAN such as ABS, SQRT, MOD, etc. The names listed next to COMMONS are the COMMON blocks that appear in the routine NAME. The routines that are listed next to CALLED BY are those routine that call NAME. A generic entry in the ROADMAP output is given by

Routine: NAME

CALLS:	SUB1, SUB2, ... ;	subroutines that NAME calls
LIB:	SUBA, SUBB, ... ;	same as above but not present in source file
COMMONS:	COM1, COM2, ... ;	common blocks used by NAME
CALLED BY:	SUBa, SUBb, ... ;	routines that call NAME

The ROADMAP output for the program ONE.FOR listed in the Appendix A is given by

```
*************************************
***   PROGRAM MAP (1 LEVEL DEEP)   ***
```

```
************************************


*** SOURCE ROUTINES ***


ROUTINE: ONE
        CALLS:     SUB1      SUB2      SUB3
        COMMONS:   PARM1     PARM2
        CALLED BY: NONE - NO ROUTINES CALL ONE

ROUTINE: SUB1
        CALLS:     SUB2
        LIB:       SUBA
        CALLED BY: ONE

ROUTINE: SUB2
        LIB:       SIN       SUBA
        COMMONS:   PARM1
        CALLED BY: ONE       SUB1

ROUTINE: SUB3
        LIB:       SUBA
        CALLED BY: ONE

ROUTINE: SUB4
        CALLED BY: NONE - NO ROUTINES CALL SUB4

*** LIBRARY ROUTINES ***


ROUTINE: SIN
        CALLED BY: SUB2

ROUTINE: SUBA
        CALLED BY: SUB1      SUB2      SUB3

************************************
***        COMMON BLOCK MAP        ***
************************************


COMMON: PARM1     SIZE =       12
   USED BY: ONE       SUB2

COMMON: PARM2     SIZE =       12
   USED BY: ONE
```

2.2 Checking Common Blocks, COMCHECK

The tool, COMCHECK, was created to identify common block variables that are changed by a subroutine. A variable has *changed* if it appears in an assignment statement on the left of an equals sign. In the CCFM/FAST consolidation project it was decided that a certain set of subroutines could access common blocks but would not be allowed to change them. The routines to be examined were responsible for calculating the right hand side of the differential equation to be solved. COMCHECK examines a FORTRAN cross-reference listing and "flags" each common block variable that was changed by a subroutine and indicates

how it was used. The usage of COMCHECK is given in below. Parameters listed in brackets are optional.

COMCHECK   The program, COMCHECK, displays information about common block variables used by a FORTRAN program. It determines which subroutines reference and change which common block variables. The file names file1 and file2 may be any valid MSDOS file name. Further, file1 may contain MSDOS wild-ard characters, '*' and/or '?' to specify a list of file names to be examined by COMCHECK. The characters '..', '.' may be used to represent the parent and current directory respectively in a path used to specify a file.

syntax:

COMCHECK   file1 [file2] [options]

file1   *required*   File containing a cross-reference listing. This file must have been generated by compiling a FORTRAN program using LAHEY 3.0 with the XREF listing option turned on.

file2   *optional*   File containing results output from this program. The default value for this parameter is the same name as file1 with the extension .CMK .

Option usage and default values are given by:

/V   Produce a table ordered by variable name. COMCHECK will not produce a table when this option is preceded with an 'N', i.e. /NV . By default this table is produced

/S   Produce a table ordered by subroutine name. COMCHECK will not produce a table when this option is preceded with an 'N', i.e. /NV . By default this table is produced.

To show how COMCHECK examines common block variable usage, consider program ONE given in Appendix A. Again suppose it is located in the file ONE.FOR. At an MSDOS prompt type

F77L   ONE

making sure that the X-REF option has been turned on. A file named ONE.LST will be created. Note that if ONE.LST was already created to use with ROADMAP then it does not have to be re-generated. Next type:

COMCHECK   ONE.LST

This command creates the file ONE.CMK which contains COMCHECK's analysis of the common blocks in program ONE.

File names used with COMCHECK, like ROADMAP, may in general be full path names containing wild card characters such as '*' or '?' and symbolic directory names such as the parent directory, '..', or the current directory, '.'. To generate a COMCHECK report for all subroutines in a directory name \PROG first compile each routine in \PROG. Next type

COMCHECK \PROG*.lst CPROG.cmk

COMCHECK will input each cross-reference listing found in the directory \PROG and output the summary to the file CPROG.CMK .

A common block variable is "written to" if it appears to the left of an equals sign(=) in an assignment statement. A common block variable is "passed" if it appears in a argument list of a subroutine. A common block variable is "used" if it is used in a subroutine in some other significant way besides "written" or "passed" Note, a common block variable is not "used" by a subroutine if it only appears in a COMMON statement or a declaration statement such as INTEGER or REAL type statements. COMCHECK flags variables that are not used by any subroutines.

In the consolidation project we were interested in knowing how routines that were called by DSOURC affected common block variables. DSOURC is the subroutine called by the ordinary differential equation (ODE) solver that calculates the right hand side of the ODE. To obtain this information we used ROADMAP to identify which routines DSOURC calls. Next we used COMCHECK to see how these routines use common block variables. After the subroutines were modified so that they no longer allowed common blocks to be written to, we re-ran COMCHECK on the new program to verify that all of the required changes were made.

The information presented by COMCHECK is arranged by variable name and by subroutine name. In the first listing given below, for each variable, COMCHECK lists each subroutine that uses that variable and how it is used. In the second listing, for each subroutine, COMCHECK lists each variable that is used by the subroutine and how it is used. The third listing gives the size in bytes of each common block found. If a common block is a different size in two different subroutines then a warning will be printed in this section.

```
        Common Block Usage by Variable Name


VARIABLE: A          COMMON: PARM2
*** NOT USED IN ANY ROUTINE ***

VARIABLE: B          COMMON: PARM2
     WRITTEN:  ONE

VARIABLE: C          COMMON: PARM2
*** NOT USED IN ANY ROUTINE ***

VARIABLE: X          COMMON: PARM1
     PASSED:  ONE      SUB2
```

```
VARIABLE: Y          COMMON: PARM1
     WRITTEN: SUB2

VARIABLE: Z          COMMON: PARM1
     WRITTEN: SUB2


Common Block Usage By Subroutine Name


ROUTINE: ONE
     WRITTEN:  B
      PASSED:  X

ROUTINE: SUB1
*** NO COMMON BLOCK VARIABLES ACCESSED BY THIS ROUTINE ***

ROUTINE: SUB2
     WRITTEN:  Y      Z
      PASSED:  X

ROUTINE: SUB3
*** NO COMMON BLOCK VARIABLES ACCESSED BY THIS ROUTINE ***

ROUTINE: SUB4
*** NO COMMON BLOCK VARIABLES ACCESSED BY THIS ROUTINE ***

COMMON BLOCK Sizes

COMMON: PARM2       SIZE:        12
COMMON: PARM1       SIZE:        12
```

The tools ROADMAP and COMCHECK were both used to produce the information contained in some of the appendices for the CCFM.VENTS Software Reference Manual and the CFAST Programmers manual. These tools can be improved by providing an option to generate the output files in a format more compatible with WordPerfect. For example, the routine name could be marked in bold and routines and common blocks contained i lists could be automatically indented. This would reduce the time required to merge the output from these tools into a report document.

2.3 Comparing Two FORTRAN Programs, FDIFF

The tool, FDIFF, was created to compare two FORTRAN programs. The CFAST suite of programs, CF_IN, CFAST and CF_PLT, had several subroutines with the same name. FDIFF was used to identify which of those routines were essentially identical. It was decided to put these routines in a common directory LIB. This makes program maintenance easier since changes need only be made in one place.

FDIFF is an enhancement of a tool, DIFFER, that was developed for the CCFM project. DIFFER is similar to many other file comparison programs in that the files to be compared are just text files. FDIFF assumes that the files to be compared are FORTRAN

programs. FDIFF therefor neglects differences that would not have any effect on the FORTRAN compilation. Some examples of non-essential differences in a FORTRAN program are changes in comment statements or spacing within a FORTRAN statement. FDIFF would consider the statement

X = 3.

the same as

X   = 3.

since the FORTRAN compiler would produce the same result in either case. The usage of FDIFF is given below. Parameters in brackets are optional

FDIFF   The program FDIFF finds the differences between two FORTRAN programs.

syntax:

FDIFF   file1 file2 [file3] [options]

file1   *required* First FORTRAN program to be compared
file2   *required* Second FORTRAN program to be compared
file3   *optional* File containing results of comparison. The default value for this parameter is the same name as file1 with the extension .DIF .

Option usage and default values are given by:

/An   Number of consecutive mis-matches that must occur before FDIFF "gives up" on comparing a routine. The default value of 'n' is 40
/F   A full report is generated. All insertions, deletions and mismatched lines are printed to file3.
/Ln   Number of lines in file1 and file2 to look ahead for a match. The default value of 'n' is 40
/Rn   This parameter determines the number of consecutive lines that must match after a mismatch occurs before the routine comparison is re-synchronized. The default value for 'n' is 3.
/S   A summary report is generated. The number of lines in each version of a routine and the number of matches are printed to file3.

To compare two FORTRAN programs consider programs ONE and TWO given in Appendix A. Type :

FDIFF   ONE.FOR TWO.FOR

This command creates the file ONE.DIF which contains FDIFF's comparison of the two

programs. FDIFF sorts each file by subroutine name so order is not important. FDIFF indicates whether a routine is present in only one of the two files. If the routine is present in both files then the two versions are compared line by line. FDIFF ignores comment statements and any blanks or tabs found in FORTRAN statements. The rational for this is that changes of this type will not alter the machine code that is generated by the compiler. The output from FDIFF upon comparing programs ONE and TWO are given by:

```
*** ROUTINE:ONE          DELETED FROM FILE:PROG1.FOR  ***

*** COMPARING: SUB1          ***
      2 LINES MATCHED IN:SUB1
      1 LINES WERE INSERTED INTO:   PROG2.FOR
            I        WRITE(6,*)' THIS LINE WAS ADDED IN PROG2'
      3 LINES MATCHED IN:SUB1

*** ROUTINE: SUB1          ***
      LINES IN    PROG1.FOR =     5
      LINES IN    PROG2.FOR =     6
      LINES THAT MATCHED =        5

*** COMPARING: SUB2          ***
      7 LINES MATCHED IN:SUB2
      NO CHANGES IN ROUTINE:SUB2

*** COMPARING: SUB3          ***
      2 LINES MATCHED IN:SUB3
      1 LINES WERE INSERTED INTO:   PROG2.FOR
            I        WRITE(6,*)' THIS IS A TEST'
      2 LINES MATCHED IN:SUB3

*** ROUTINE: SUB3          ***
      LINES IN    PROG1.FOR =     4
      LINES IN    PROG2.FOR =     5
      LINES THAT MATCHED =        4
*** ROUTINE:SUB4          DELETED FROM FILE:PROG1.FOR  ***
*** ROUTINE:SUB5          INSERTED INTO FILE:PROG2.FOR ***
*** ROUTINE:SUB5          INSERTED INTO FILE:PROG2.FOR ***
```

3. INTERFACE MODULES

The software modules documented in this section perform various utility functions required by tools presented in Section 2 and fire models such as CFAST and CCFM.VENTS. Some examples of the use of an interface in these programs are routines for performing screen input and output, parsing character strings, manipulating character strings, managing blocks of memory and handling files. These software building blocks allow productivity increases by re-using code to perform similar functions.

3.1 Screen Input and Output

The routines described in this section incorporate only the most primitive elements of windowing. Further they are text based. There are two reasons for this. First, the idea of portability is important. These concepts have been used in the models described earlier on a very wide range of computers. Initial development was on a Concurrent 3200 series computer. This was followed by a Cyber 855, an Apollo workstation, and currently the MSDOS computers which will be discussed here. Others have taken these routines and put equivalent modules on the VAX series of computers. The point is that FORTRAN is the most widely supported language for scientific computing. This type of windowing is quite useful for developing and using programs, and these concepts can be utilized in most computing environments.

The second point is that simple but elegant displays of information can be extremely useful for conveying ideas. In almost all cases, this can be accomplished with the minimum of tools. Extremely complex modules usually only add headaches and not value. Further, with portability one of our overriding concerns, simplicity is of the essence. An example of the use of these routines is given at the end of the section. The code for this example is shown in Appendix B.

3.1.1 Initialization, CSPACE

Initialization of the screen input/output routines consists of determining the computer environment that one is using. This involves determining the type of screen (dimensions or pixel resolution), directory structure and so forth. The routine which does this for the screen input and output is **CSPACE**. An example will be given later.

The primary requirement to use the services which follow is that the terminal, or graphics adaptor, be able to switch modes while it is being used. The basic command set is the ability to switch to and from reverse video, and move and locate the cursor on the screen. Color is very useful, but not essential in any of the routines which follow.

3.1.2 Windows, WINDOW

The most important advantage of text based input and output is that video screens are reasonably well defined. In general there are 80 characters per line, and 23 to 25 lines per screen. There are many variants on this general theme, but by sticking to these limitations, portability is generally assured. The routine is **WINDOW** and is invoked by

```
CALL WINDOW (TOP, LEFT, BOTTOM, RIGHT, FOREGROUND, BACKGROUND)
```

All four arguments are integers, and define the upper left and lower right corners of the window. For a standard screen, these numbers can vary from 0 to 24. There are, however, many variations, with some top left corners beginning at one, and some screens with only 23 lines.

3.1.3 Scrolling, SCROLLU and SCROLLD

A window can be moved up or down by one or more rows. Indeed, one might consider left or right scrolling. In some cases we do that, but it happens seldom enough that general purpose code is not worthwhile. Also, there is a mechanism built into most display adapters and terminals to enable very fast scrolling in the vertical direction. Thus there is a much bigger gain in direct access to the video hardware for this case.

The routines for scrolling up and down are SCROLLU and SCROLLD respectively. The protocol is

```
CALL SCROLLU (TOP, LEFT, BOTTOM, RIGHT, BACKGROUND, FOREGROUND)
```

and

```
CALL SCROLLD (TOP, LEFT, BOTTOM, RIGHT, BACKGROUND, FOREGROUND)
```

All arguments are integers. The first four are analogous to the arguments for WINDOW. The last two specify the background and foreground colors to use in subsequent text operations.

3.1.4 Cursor visibility, CURSON and CURSOF

There are also complementary routines for turning the cursor on and off. They are CURSON and CURSOF. Their use should be obvious.

3.1.5 Output to the screen, MESSNR

The routine MESSNR writes a character variable to the screen. The present implementation uses the PC BIOS routines, but there are equivalent direct screen writes for display adapters, and of course serial output over RS232 lines can be done in a similar manner. In our case, the reason for doing it through the BIOS interrupts is the portability issue again. Normally, if one were concerned only with speed, and the target were a specific computer, then direct screen writes are considerable faster.

The protocol is

```
CALL MESSNR (MESSAGE, LENGTH).
```

The MESSAGE is a character variable, and LENGTH is an integer. An alternative would be to require the C protocol which terminates a string with a null (hex 0). This the would not require an explicit length parameter. Our formation is more general, however, and includes it as a subset.

3.1.6 Cursor positioning, CHRLOC and CHRMOV

CHRLOC finds the position of the cursor and CHRMOV moves the cursor to a specific location. There is one part of this that is tricky. Earlier it was stated that most character based displays operate with 80 columns and 23 to 25 lines. There is no safeguard for 40 column displays within the routines themselves. In the case of PC's, the BIOS will protect the user from harming the system, and similar checks are done for terminals. However, such considerations do come into play in designing screens. All displays which allow cursor manipulation also have a provision for ascertaining the size of the screen. This service is not provided here, but if dealing with a wide variety of displays, then this should be a consideration.

The protocol is

```
CALL CHRLOC (X, Y) AND CALL CHRMOV (X, Y)
```

In both cases, X and Y are integers, and generally range from 0 to 24 and 0 to 79 respectively. Once again, there are many variations on this theme, such as beginning at one, or ending at 25, 23 or some other length.

3.1.7 Inverting text, BARCODE

It is sometimes useful to be able to highlight text. This is most easily done simply by inverting the colors in the attribute byte of a display. The fact that the arrangement of the bytes in display memory varies widely from system to system is the biggest argument for letting the system handle functions such as writing to the screen or inverting text. Otherwise, one is faced with the problem of handling each case separately.

The protocol is

```
CALL BARCODE (FLAG, ROW, COLUMN, LENGTH OF THE BAR)
```

FLAG is not used in this implementation. Originally it was used to set the direction of the inversion, whether on or off. BARCODE simply acts as a toggle for the inverse text. ROW, COLUMN specify the starting position and LENGTH is the number of characters to invert. One must be cognizant of the offset discussed above for CHRLOC and CHRMOV.

3.1.8 Saving and restoring screens, VSTOW and VSHOW.

One of the many tricks in windowing systems is the ability to insert some text, a help screen for example, and then restore the text. These services are available with the routines VSTOW and VSHOW. The former saves the contents of the display memory in a buffer, and the latter restores the screen. The screen saves are stacked, and then restored in reverse order. The use is demonstrated in the routine SWINDOW. Within that routine is the logic for actually doing the screen input and output, once again through the BIOS routines.

```
CALL VSTOW (BUFFER, ROW, COLUMN, LENGTH)
```

and

```
CALL VSHOW (BUFFER, ROW, COLUMN, LENGTH).
```

The arguments are integers. The buffer must be big enough to contain the size of the screen being saved. The low level implementation of this is very hardware specific in that the actual form of the screen data determines the size of the buffer. For example, in our present example, the buffer is a sixteen bit integer array, since each character on the screen has only an attribute and character byte associated with it. In a more general system, there might be information on the relative intensity of the red, green and blue video guns, whether the character is visible, and so on.

There is no requirement that the data go back to the place from which it came, so this has the side effect of allowing fast moves of blocks of data. In a general sense, these routines are memory management routines. However, screen information is a very special type of memory, and conceptually can most usefully thought of as distinct from normal computer RAM. ROW and COLUMN specify the starting position, and LENGTH is the number of characters to be saved.

3.1.9 Saving and restoring the palette registers.

Most display devices go through a look up table to translate attribute bytes into colors which then appear on the screen. There are many advantages to this scheme, not the least of

which is the reduction in the cost by reducing the number of chips necessary to display information. The routines are EGAGET, EGAPUT and SETPAL. The procedure calls are

```
CALL EGAGET (TABLE, L1, L2)
CALL EGAPUT (TABLE)
```

and

```
CALL SETPAL (TABLE)
```

where TABLE is the appropriate table for the hardware of interest. In the case of the PC, this is an integer array of length 17 whose size is two bytes per entry. Another example would be the Lexidata 3700 which uses a table whose size is 4×256 of 16 bit integers.

An example of using these routines is shown in the first part of the program in Appendix B. This example opens a window, moves the cursor to various locations and inserts text. The result of the screen up to the first request for keyboard input is

```
┌─────────────────────────────────────────┐
│TWINDOW v 1.0                             │
│05/06/90                                  │
│                                          │
│   »Main Heading                          │
│                                          │
│    Sub Heading                           │
│                                          │
│    Normal Text                           │
│                                          │
│    Protected Text                        │
│                                          │
│                                          │
│                                          │
│                                          │
│                                          │
│                                          │
│   ████████████████████████               │
│                              ^X          │
└─────────────────────────────────────────┘
```
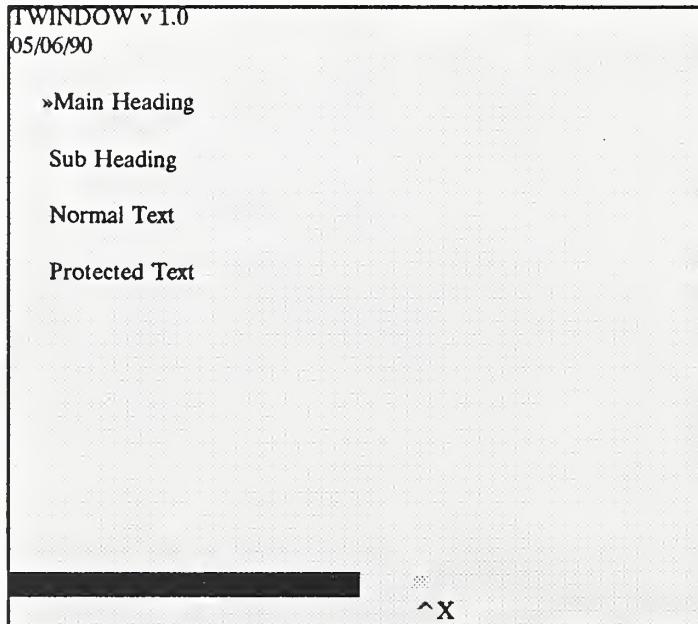
Example of windowing showing text and cursor movement

This example is continued in section 3.5.5.

3.2 Parsing Text

One aspect of a user-friendly program is to allow free form input; that is, not requiring

input to the program to occur in any particular column or columns. A program then processes the input by parsing a line of text into a series of tokens. A *token* is a group of text that "stands on its own". For example, in the FORTRAN statement

XYZ = ABC + DEF

the compiler would consider the following strings in quotes to be tokens 'XYZ', 'ABC', 'DEF', '=', '+'. In our applications a token can usually be thought of as a contiguous group of non-blank characters.

The parsing routines documented in this section each process different types of text. The routine, PARSE, processes a general line of text. The routine, FPARSE, parses a file name with or without a path. The routine, PPARSE, parses a command line and records information about options delimited with a '-' or a '/'.

3.2.1 Parsing a General Line of Text, PARSE

The routine, PARSE, determines the beginning and ending of each token contained in a line of text. A blank or a comma is used to separate tokens. The token boundaries found by PARSE are used by the calling program to identify and handle input data.

The protocol is

```
CALL PARSE(LINE,SB,SE,NTKS)
```

where

LINE is a character variable containing the line of text to be parsed;
SB,SE are integer arrays containing the beginning and ending of each token, the I'th token contained in LINE would be given by LINE(SB(I):SE(I)), if the I'th token is absent from LINE then SB(I)=0;
NTKS is the number of tokens found.

3.2.2 Parsing a File Name, FPARSE

Full file names parsed by FPARSE have the form X:\dir1\ ... \dirN\file where X is a valid drive and dir1, ... , dirN are valid directory names and file is a valid file name. The software tools described in Section 2 accepts full file names as input. This routine determines the local file name in a convenient manner. It would be FILE(SB(N):SE(N)), if the I'th component of the full file name is absent then SB(I) =0

The protocol is

```
CALL FPARSE(FILE,SB,SE,N)
```

where

FILE is a character variable containing the file name, possibly including the full path name;
SB,SE are integer arrays containing the beginning and ending of each file name component, where a component may be a drive, directory or file name. The I'th component of the full file name is given by FILE(SB(I):SE(I)). If the I'th component is absent then SB(I) = 0;
N is the number of components that make up the file name.

3.2.3 Parsing a Command Line with Options, PPARSE

The routine, PPARSE is used to parse a string of text contained on the command line. It considers two types of tokens, parameters and options. An option modifies how a parameter is treated by the program. For example, in the tool FDIFF, the /F option causes FDIFF to generate a full report to the output file rather than a summary report. An option begins with either a '-' or a '/' character.

The protocol is

```
CALL PPARSE(LINE,SBPARM,SEPARM,NPARM,SBOPT,SEOPT,AFTPRM,NOPT)
```

where

LINE is a character variable containing the command line to be parsed;
SBPARM and SEPARM
  are integer arrays containing the beginning and ending of each parameter. The I'th parameter is given by LINE(SBPARM(I):SEPARM(I)). If the I'th parameter is absent then SB(I)=0
NPARM is the number of parameters;
SBOPT and SEOPT
  are integer arrays containing the beginning and ending of each option contained on the command line;
AFTPRM is an integer array identifying which parameter the option follows;
NOPT is the number of option tokens on the command line.

3.3 Character Manipulation and Conversion

3.3.1 Identifying Alphabetic Characters, ALPHA

The logical function, ALPHA, returns .true. if its input is an alphabetic character (a character between 'A' and 'Z' or 'a' and 'z') and .false. otherwise.

The protocol is

```
LALPHA = ALPHA(C)
```

where

C   is a character variable of length 1;
ALPHA   is a logical function.

3.3.2 Finding the Length of a Character String, LENGTH

The integer function, LENGTH, returns the position of the last non-blank character in a character string. This is different than the FORTRAN function, LEN. LEN returns the length of the space allocated to a character string. To clarify the difference between LEN and LENGTH consider the following two FORTRAN statements.

```
CHARACTER*80 LINE
LINE='ABC'
```

LEN(LINE) = 80 which is the space allocated to LINE while LENGTH(LINE) = 3 which is the amount of information actually contained in LINE.

The protocol is

```
ITEMP = LENGTH(LINE)
```

where

LINE   is a character variable of arbitrary size;
LENGTH   is an integer function which determines the column number containing the last non-blank character in LINE.

3.3.3 First non-blank character, NOBLNK

This integer function, NOBLNK, returns the column of a character string containing the first nonblank character and a zero if the character string is completely blank. This function is the reverse of LENGTH. NOBLNK(LINE) and LENGTH(LINE) point to the first and last non-blank characters in LINE.

The protocol is

```
ITEMP = NOBLNK(LINE)
```

where

LINE   is a character variable of arbitrary size;
NOBLNK   is an integer function which calculates the column of LINE that contains the first non-blank character.

3.3.4 Identify a Numerical Character, NUM

The logical function, NUM, returns .true. if the input is a number between 0 and 9 and returns .false. otherwise.

The protocol is

```
LTEMP = NUM(CHAR)
```

where

CHAR   is a character variable of size 1;
NUM   is a logical function which returns .true. is CHAR is a number and returns .false. otherwise.

3.3.5 Changing the Case of Character(s) UPPER, TOUPPER, TOLOWER

The routine, UPPER, converts characters in the input character variable, LINFRM, to upper case. TOUPPER is the equivalent of UPPER in FUNCTION form. TOLOWER performs the opposite function, converting a character to lower case. The functional form of TOLOWER is the same as TOUPPER. UPPER converts a character string of arbitrary size to upper case while TOUPPER only converts a single character.

The protocol for UPPER is

```
CALL UPPER(LINFRM,LINTO)
```

where

LINFRM   is a character variable containing the string to be converted;
LINTO   is a character variable containing the converted string.

Note: LINFRM and LINTO may be the same variable in the calling routine. The character conversion is done "in place".

The protocol for TOUPPER and TOLOWER is

```
CHARACTER*1 TOUPPER, TOLOWER, CHARACTER
        .
        .
ICHAR = TOUPPER (CHARACTER)
```

where

CHARACTER is the character of size 1 to be converted,
ICHAR   is the converted character again of size 1.

3.3.6 Finding character strings in a text line, SSTRNG

SSTRNG is used to find continuous strings of text within a character string. It serves a purpose similar to PARSE in section 3.2.1, but looks at the tokens one at a time. This is useful for context sensitive decoding of a string. The procedure is

SSTRNG (string, start, count, first, last, valid).

String is a character string, start is the offset within the string to begin the search, count is the total number of characters to search, first is the offset of the first nonblank character, last is the offset of the last continuous nonblank character. These latter four are integers. Valid is a logical variable, to indicate whether any valid string was found. Any nonblank character is valid data, so control characters count in this sense.

3.4 Memory Management

It is often not known when writing a program how much memory will be required. For example, ROADMAP requires memory to store ancestors and descendants of each subroutine. Some subroutines could have many more ancestors than others. Statically dimensioning FORTRAN arrays so that each array could contain a maximum number of ancestors would be inefficient, since many subroutines would not require the space. A memory manager then is a means to allocate memory as needed on the fly.

The memory module is a collection of routines written in standard Fortran that provide a means for allocating and de-allocating blocks of memory. These FORTRAN allocation/de-allocation routines are similar to the capabilities provided with C and Pascal. This allocation process is not truly dynamic, for all memory assigned by the memory module comes from the common block /MEMRY/. This was done to allow this software module to run on any machine that supports standard FORTRAN 77.

3.4.1 Copy blocks of memory, CPYPTR

The routine, CPYPTR, copies a block of memory to another portion of memory and returns a pointer to the new block.

The protocol is

```
CALL CPYPTR(OLDPTR,TYPE,NEWPTR,NEWSIZ,ERCODE)
```

where

OLDPTR   is an integer pointing to a memory block that is to be copied,
TYPE   is an integer specifying the type of block (1=integer, 2=floating point)
NEWPTR   is an integer pointing to the re-sized memory block,
NEWSIZ   is an integer specifying the size of the new block,
ERCODE   is an integer returning the error code, 0, if all is o.k.

3.4.2 Validity of a pointer to memory, DCODE

The routine, DCODE, determines whether a pointer to a block of memory is valid. This routine is used by other memory management routines to insure that they were passed valid memory pointers.

The protocol is

```
LCODE = DCODE(JDATA)
```

where

JDATA   is the first array element of a memory block to be decoded.
DCODE   is a logical function return .true. if the block is valid and .false. otherwise.

3.4.3 Delete a block of memory, DELPTR

The routine, DELPTR deletes a block of memory.

The protocol is

```
CALL DELPTR(PTR,TYPE,ERCODE)
```

where

PTR   is an integer pointing to a memory block that is to be deleted,
TYPE   is an integer specifying the type of block (1=integer, 2=floating point)
ERCODE   is an integer returning the error code, 0, if all is o.k.

3.4.4 Allocate a block of memory, GETPTR

The routine, GETPTR, allocates a block of memory of a given size and type. The valid types are integer and floating point. This is similar to the functions MALLOC and CALLOC used in the C programming language.

The protocol is

```
CALL GETPTR(PTR,SIZE,TYPE,ERCODE)
```

where

PTR   is an integer pointing to a memory block that is to be allocated,
SIZE   is the size of the block to be allocated,
TYPE   is an integer specifying the type of block (1=integer, 2=floating point)
ERCODE   is an integer returning the error code, 0, if all is o.k.

3.4.5 Size of a block of memory, GETSIZ

The routine, GETSIZ, determines the size of a block of memory. This routine requires a pointer to the block and its type (integer, real or double precision). The initialization routine INITMM specifies how many integers correspond to a unit of both single and double precision memory.

The protocol is

```
CALL GETSIZ(PTR,TYPE,SIZE)
```

where

PTR   is an integer pointing to a memory block whose size is to be found,
TYPE   is an integer specifying the type of block (1=integer, 2=floating point)
SIZE   is the size of the block,

3.4.6 Initializing the Memory Manager, INITMM

This routine initializes the memory manager by defining a first and last block which are used internally by the memory manager and cannot be deleted

The protocol is

```
CALL INITMM
```

3.4.7 Encoding a Memory Block, NCODE

The routine, NCODE, encodes a block of memory so that the block can be checked later for validity i.e. when deleting a block of memory we make sure that the block is valid.

The protocol is

```
CALL NCODE(JDATA)
```

where

JDATA   is the first array element of a memory block to be decoded.

3.4.8 Changing the Size of a Memory Block, RESIZ

The routine, RESIZ, resizes a block of memory. If there is not enough room where the block is located it will allocate another block and copy the old block to the newly allocated block.

The protocol is

```
CALL RESIZ(PTR,NEWSIZ,TYPE,ERCODE)
```

where

PTR   is an integer pointing to a memory block that is to be re-sized,
NEWSIZE   is the new size of the block,
TYPE   is an integer specifying the type of block (1=integer, 2=floating point)
ERCODE   is an integer returning the error code, 0, if all is o.k.

3.5 File Handling

The routines shield the programs from details of the underlying operating system. They all perform fairly obvious, and often used functions. Most programs (and programmers) do not care about the details of how these operations are carried out. Thus we provide generic file functions, whose details can be modified to suit each language or operating system as necessary.

3.5.1 Determining Whether a File Name is Valid, LEGAL

The logical function, LEGAL, returns .true. if file is a legal file name and .false. other-

wise.

The protocol is

```
LFILE = LEGAL(FILE)
```

where

FILE   is a character variable containing a candidate file name
LEGAL   is true or false depending on whether FILE is a valid file name or not

3.5.2 Opening a File, OPENFL

The routine, OPENFL, provides a central location where a file can be opened for input or output depending on the value of the parameter, ICODE. It also does some rudimentary error handling. Before opening the file for input, OPENFL checks to see if the file exists. If the file does not exist, then a non-zero error code is returned. Opening files in a central location makes it easier to port program to other computers since I/O requirements, specific forms of key words for example, vary from one computer to the next.

The protocol is

```
CALL OPENFL(IUNIT,PATH,FFILE,FTEMP,ICODE,IERR)
```

where

IUNIT   is the FORTRAN unit number used to reference the file;
PATH   a character variable containing the path name of the file to be
              opened, this parameter may be blank;
FFILE   a character variable containing the file name, OPENFL
              attempts to open the file PATH//FFILE;
FTEMP   a character scratch variable;
ICODE   ICODE=0 for input, ICODE=1 for output;
IERR   IERR=0 if all went o.k.

3.5.3 Creating a New File, NEWFIL

This routine was written to insure uniform behavior of programs that open files for output when the file already exists. This routine tries to open a file specified by FNAME. IF the file already exists then NEWFIL deletes it and opens a clean copy.

The protocol is

```
      CALL NEWFIL(IUNIT,FNAME)
```

where

IUNIT   is the FORTRAN unit number used to reference the file and

FNAME   is a character variable containing the file to be opened.

3.5.4 Speeding Up File I/O, DABUFR

The tool, FDIFF, described in Section 2 uses the routine DABUFR to reduce the time required to perform disk I/O. This routine is written in standard FORTRAN but will work best when the size of the disk buffer, CBUF, is a multiple of the sector size of the hard disk used which is 512 bytes for MSDOS formatted disks. Note that the record size of CBUF is determined by the calling routine not by DABUFR.

FDIFF uses direct access files to store the files it is comparing. It uses direct access files rather than sequential files so that it can later access any subroutine in any order. DABUFR buffers input and output by using the fact that when a line of text needed by a program it is likely that other lines near this one will also be required. DABUFR stores many lines of text into one record. The calling routine requests a line of text from DABUFR. If the line is in the buffer, CBUF, then it is copied to the variable, CREC. If the line is not in CBUF then DABUFR has to read from disk to get the correct record.

The protocol is

```
      CALL DABUFR(IODA,CBUF,IBUF,NBUFS,CREC,IREC,INOUT,IERR,IWORK)
```

where

IODA   the unit number used to reference the direct access file being used;

CBUF   a character variable containing many lines of text. Each line of text must be the same size as CREC;

NBUFS the total number of 'CBUF' records contained in the file referenced by IODA;

CREC   a character variable containing the line of text that the calling routine retrieved from the direct access file (if INOUT=0) or wrote to the direct access file (if INOUT = 1);

IREC   the record number;

INOUT INOUT = 0 means that the calling routine is writing to disk, INOUT = 1 means that the calling routine wishes to read from disk;

IERR   IERR = 0 if all is o.k.;

IWORK An integer array of size 4, which contains pointer information.; IWORK(1) must be zero on the first call to DABUFR; since this routine can be used for many buffers, a separate CBUF and IWORK are required for each.

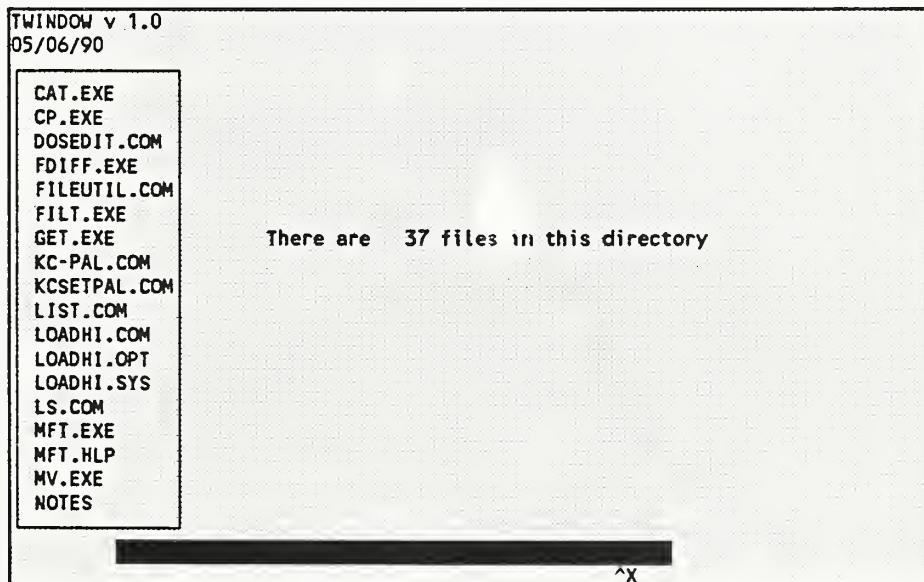### 3.5.5 Finding files, FFILE, FILEFRST, FILENEXT and FILELAST

This routine looks up a file based on the path given to it. No assignment is done, but rather a list of valid file names is returned. Its use is something like the INQUIRE statement from FORTRAN, but it will use a template to search for a list of files. The actual implementation is platform dependent, of course. In this case, the routine FFILE makes calls to the routines FILEFRST, FILENEXT and FILELAST which in turn use the MSDOS interrupt services. This is much faster than a general search of file names. For MSDOS systems it is *extremely* important that FILELAST be called after FILEFRST is called. There is a po inter in the task for buffered input and output. This gets exchanged while doing the file find function. It must be restored, or the system will crash randomly. The space provided by the calling program may not be sufficient for the general case that the operating system deals with. The procedure calls are

```
CALL FFILE (FILE, NRTN)
CALL FILEFRST (FILEN, BUFFER, NRTN)
CALL FILELAST
```

FILE, FILEN and BUFFER are character variables. The lengths must be at least

FILE	12
FILEN	43
BUFFER	128.

An example of its usage is shown in the continuation of the program in Appendix B.

```
TWINDOW v 1.0
05/06/90

 CAT.EXE
 CP.EXE
 DOSEDIT.COM
 FDIFF.EXE
 FILEUTIL.COM
 FILT.EXE
 GET.EXE            There are   37 files in this directory
 KC-PAL.COM
 KCSETPAL.COM
 LIST.COM
 LOADHI.COM
 LOADHI.OPT
 LOADHI.SYS
 LS.COM
 MFT.EXE
 MFT.HLP
 MV.EXE
 NOTES

                                                      ^X
```

Starting at the point in section 3.1 which showed the example of a window with text, press the <PgDn> key to continue. This invokes the FFILE procedure, which lists all of the files in the current directory, and puts this list on the screen. It shows also the use of the vstow functions to save sections of the screen. Pressing escape restores the screen. The cursor keys can be used to scroll up and down through the list of files, if there are more than 18, the number that can be shown at one time.

APPENDIX A Sample Programs for Development Tools

The following two example programs were used to illustrate the use of ROADMAP, FDIFF and ROADMAP.

```
        PROGRAM ONE
C
C*** THIS PROGRAM DOES NOT DO ANYTHING USEFUL
C    IT IS USED AS AN EXAMPLE TO ILLUSTRATE THE USE
C    OF ROADMAP AND COMCHECK.  IT IS ALSO COMPARED WITH
C    PROGRAM TWO TO ILLUSTRATE THE USE OF FDIFF
C
        COMMON /PARM1/X,Y,Z
        COMMON /PARM2/A,B,C
        CALL SUB1
        CALL SUB2(X)
C
C*** COMCHECK SHOULD SHOW THAT B IS WRITTEN
C
        B = 1.
        CALL SUB3
        STOP
        END
        SUBROUTINE SUB1
        CALL SUBA
        CALL SUB2
        RETURN
        END
        SUBROUTINE SUB2
        COMMON /PARM1/X,Y,Z
C
C*** COMCHECK WILL SHOW THAT BOTH Y AND Z ARE WRITTEN TO
C    AND THAT X IS REFERENCED IN A SUBROUTINE
C
        Z = 1.
C
C*** ROADMAP WILL SHOW THE SIN FUNCTION AS A LIBRARY ROUTINE
C    SINCE ITS SOURCE IS NOT PRESENT IN THIS FILE
C
        Y = SIN(X)
        CALL SUBA
        RETURN
        END
        SUBROUTINE SUB3
        CALL SUBA
        RETURN
        END
        SUBROUTINE SUB4
        RETURN
        END

        PROGRAM TWO
C
C*** THIS PROGRAM DOES NOT DO ANYTHING USEFUL
C    IT IS USED AS AN EXAMPLE TO ILLUSTRATE THE USE
C    OF ROADMAP AND COMCHECK.  IT IS ALSO COMPARED WITH
```

```
C     PROGRAM ONE TO ILLUSTRATE THE USE OF FDIFF
C
        COMMON /PARM1/X,Y,Z
        COMMON /PARM2/A,B,C
        CALL SUB1
        CALL SUB2
C
C*** THESE COMMENTS ARE DIFFERENT BUT FDIFF DOESN'T CARE.  IT
C    IGNORES COMMENT STATEMENTS AND ANY BLANKS OR TABS CONTAINED
C    IN A FORTRAN STATEMENT
C
        B = 1.
        CALL SUB3
        STOP
        END
        SUBROUTINE SUB2
        COMMON /PARM1/X,Y,Z
C
C*** COMCHECK WILL SHOW THAT BOTH Y AND Z ARE WRITTEN TO
C    AND THAT X IS REFERENCED IN A SUBROUTINE
C
        Z = 1.
C
C*** ROADMAP WILL SHOW THE SIN FUNCTION AS A LIBRARY ROUTINE
C    SINCE ITS SOURCE IS NOT PRESENT IN THIS FILE
C
        Y = SIN(X)
        CALL SUBA
        RETURN
        END
        SUBROUTINE SUB3
        CALL SUBA
       WRITE(6,*)' THIS IS A TEST'
        RETURN
C
C*** NOTE THAT SUBROUTINE SUB4 WAS REMOVED FROM PROGRAM TWO
        END
        SUBROUTINE SUB1
        CALL SUBA
        WRITE(6,*)' THIS LINE WAS ADDED IN PROG2'
        CALL SUB2
        RETURN
        END
        SUBROUTINE SUB5
C
C*** NOTE THAT SUBROUTINE SUB5 WAS ADDED TO PROGRAM TWO
C
        RETURN
        END
```

APPENDIX B Sample Programs for Software Modules

The following program is used to show an implementation of the routines in section 3.1.

```
      PROGRAM TESTWIND

C     MAIN DRIVER PROGRAM FOR ILLUSTRATION OF WINDOWING SCHEME

      CHARACTER BLOCK*1(4), POINT*1, POINTU, DATOUT*8, FILEN*13
      INTEGER RWHERE(4), CWHERE(4), SPKEY, ICSETV(11), VERSION
      LOGICAL SPECIAL

C     PASS THE COLORS AROUND

      COMMON /TW/ICHDR,ICSUB,ICTXT,ICPRO,ICMSG,ICHLP,ICBG,ICMBG,
    . ICHBG, ICEBG, ICEMS
      EQUIVALENCE (ICHDR,ICSETV)

C     VERSION = 100*MAIN VERSION NUMBER + SUB VERSION NUMBER

      DATA BLOCK/4*Z'DB'/, POINT/Z'10'/, RWHERE/2,4,6,8/
      DATA CWHERE/6,6,6,6/, POINTU/Z'18'/
      DATA ICSETV/14,5,15,3,10,15,0,1,1,7,12/

C     INITIALIZATION

      IW = 1
      VERSION = 100
      CALL DATE (DATOUT)
      CALL CSPACE
      CALL CURSOF

      ICBGP = ICBG
  104 CALL HEADING (VERSION, DATOUT)
      CALL WINDOW (16, 0, 24, 79, ICBGP, ICTXT)
      DO 103 I = 1, 16
  103 CALL MESSNS (21, 6+I*3, BLOCK, 3, ICBG, I-1)
  102 CALL WINDOW (1, 0, 16, 79, ICBG, ICBG)

1     CALL MESSNS (2,  5, 'Main Heading', 12, ICBG, ICHDR)
      CALL MESSNS (4,  5, 'Sub Heading', 11, ICBG, ICSUB)
      CALL MESSNS (6,  5, 'Normal Text', 11, ICBG, ICTXT)
      CALL MESSNS (8,  5, 'Protected Text', 14, ICBG, ICPRO)
C
  100 IR = RWHERE(IW)
      IC = CWHERE(IW)
      CALL MESSNS (IR, IC-2, POINT, 1, ICBG, ICTXT)
      CALL MESSNS (22, 10+3*ICFORE, POINTU, 1, ICBGP, ICBGP)
      ICFORE  = ICSETV(IW)
      CALL MESSNS (22, 10+3*ICFORE, POINTU, 1, ICBGP, ICTXT)
  101 CALL NPUTKB (SPECIAL, SPKEY, 'PgDn to show a list of files', 28)
      IF (.NOT.SPECIAL) GO TO 101
      IF (SPKEY.EQ.13) THEN
          CALL MESSNS (IR, IC-2, POINT, 1, ICBG, ICBG)
          IW = IW - 1
          IF (IW.LT.1) IW = 4
          GO TO 100
      ELSE IF (SPKEY.EQ.18) THEN
```

```
            CALL MESSNS (IR, IC-2, POINT, 1, ICBG, ICBG)
            IW = IW + 1
            IW = MOD(IW-1,4) + 1
            GO TO 100
         ELSE IF (SPKEY.EQ.15) THEN
            CALL MESSNS (22, 10+3*ICFORE, POINTU, 1, ICBGP, ICBGP)
            ICFORE = ICFORE - 1
            IF (ICFORE.LT.0) ICFORE = 15
            ICSETV(IW) = ICFORE
         ELSE IF (SPKEY.EQ.16) THEN
            CALL MESSNS (22, 10+3*ICFORE, POINTU, 1, ICBGP, ICBGP)
            ICFORE = ICFORE + 1
            ICFORE = MOD(ICFORE, 16)
            ICSETV(IW) = ICFORE
         ELSE IF (SPKEY.EQ.11) THEN
            CALL CSPACE
            STOP
         ELSE IF (SPKEY.EQ.19) THEN
            FILEN = '*.*'
            CALL SWINDOW (2, 2, 24, 57)
            CALL FFILE (FILEN, NRTN)
            CALL RWINDOW
         ENDIF
         GO TO 1
         END

         SUBROUTINE FFILE (FILEN, NRTN)

C     THIS ROUTINE USES THE DOS INTERRUPT SERVICE TO SEARCH FOR A LIST
C     OF FILES.  THE DATA STRUCTURE IS IN LFILE, THE LIST OF FILES IN
C     FILEX AND THE PATH IN DATAPATH.

C     IF THERE ARE MORE THAN 16 FILES IN THE DIRECTORY, THEN YOU CAN SCOLL

         PARAMETER (MAXFIL=200)
         LOGICAL SPECIAL, VALID
         CHARACTER LBUF*128, LFILE(43)*1, FILEX(MAXFIL)*13, FILEY*13,
        . FILEN*13
         INTEGER FILECNT, SPKEY, WHERE, QFILE, FIRST, COUNT
         EQUIVALENCE (LFILE(31),FILEY)

         COMMON /TW/ ICHDR,ICSUB,ICTXT,ICPRO,ICMSG,ICHLP,ICBG,ICMBG,
        . ICHBG, ICEBG, ICEMS
C
         NRTN = 0
         FILECNT = 0
         WHERE = 2
         LBUF = FILEN//CHAR(0)
         FILEY = ' '
         CALL FILEFRST(LFILE, LBUF, NRT)    .
         IF (NRT.EQ.0) THEN
            NRTN = 0
            CALL FILELAST
            RETURN
         ENDIF
         FILECNT = 1
         FILEX(1) = ' '
         CALL SSTRNG (FILEY, 13, 1, FIRST, LAST, VALID)
         COUNT = LAST - FIRST + 1
         FILEX(1)(1:COUNT) = FILEY(FIRST:LAST)

2        FILEY = ' '
         CALL FILENEXT(NRT)
         IF (NRT.GT.0) THEN
```

```
                FILECNT = FILECNT + 1
                FILEX(FILECNT)(1:13) = FILEY
                IF (FILECNT.LT.MAXFIL) GO TO 2
            ENDIF
        3 CALL FILELAST

          WRITE(LBUF,1) FILECNT
1         FORMAT('There are ',I4,' files in this directory')
          CALL MESSNS (8, 20, LBUF, 38, ICMBG, ICTXT)
          LCOUNT = MIN(18, FILECNT)
          LCOUNT_1 = LCOUNT + 1
          CALL WINDOW (2, 2, LCOUNT_1, 14, ICHBG, ICTXT)

C     SORT BY NAME AND EXTENSION

          CALL FILESORT (FILEX, FILECNT)

C     AND LIST IN A WINDOW

          DO 5 I = 1, LCOUNT
          CALL CHRMOV (I+1,2)
5         CALL MESSNR (FILEX(I), 13)
          CALL BARCODE (1, WHERE, 2, 13)
          QFILE = 1

      91 CALL NPUTKB (SPECIAL, SPKEY, 'This shows overlapping windows', 30)
          IF (SPECIAL) GO TO 92
          FILEN = FILEX(QFILE)
          NRTN = QFILE
          RETURN

      92 IF (SPKEY.EQ.18) THEN
              IF (WHERE.LT.LCOUNT_1) THEN
                  CALL BARCODE (0, WHERE, 2, 13)
                  WHERE = WHERE + 1
                  QFILE = QFILE + 1
                  CALL BARCODE (1, WHERE, 2, 13)
              ELSE
                  IF (QFILE.LT.FILECNT) THEN
                      CALL BARCODE (0, WHERE, 2, 13)
                      CALL SCROLLU (2, 2, LCOUNT_1, 14, ICHBG, ICTXT)
                      QFILE = QFILE + 1
                      CALL CHRMOV (LCOUNT_1, 2)
                      CALL MESSNR (FILEX(QFILE), 13)
                      CALL BARCODE (1, WHERE, 2, 13)
                  ENDIF
              ENDIF
          ELSE IF (SPKEY.EQ.13) THEN
              IF (WHERE.GT.2) THEN
                  CALL BARCODE (0, WHERE, 2, 13)
                  WHERE = WHERE - 1
                  QFILE = QFILE - 1
                  CALL BARCODE (1, WHERE, 2, 13)
              ELSE
                  IF (QFILE.GT.1) THEN
                      CALL BARCODE (0, WHERE, 2, 13)
                      CALL SCROLLD (2, 2, LCOUNT_1, 14, ICHBG, ICTXT)
                      QFILE = QFILE - 1
                      CALL CHRMOV (2, 2)
                      CALL MESSNR (FILEX(QFILE), 13)
                      CALL BARCODE (1, WHERE, 2, 13)
                  ENDIF
              ENDIF
          ELSE IF (SPKEY.EQ.1) THEN
```

```
                  FILEN = FILEX(QFILE)
                  NRTN = QFILE
                  RETURN
              ELSE IF (SPKEY.EQ.11) THEN
                  NRTN = 0
                  FILEN = ' '
                  RETURN
              ENDIF
              GO TO 91
              END

              SUBROUTINE HEADING (VERSION, DATOUT)

C      DISPLAY THE HEADING

              INTEGER VERSION
              CHARACTER DATOUT*8, LBUF*128

              COMMON /TW/ICHDR,ICSUB,ICTXT,ICPRO,ICMSG,ICHLP,ICBG,ICMBG,
           .  ICHBG, ICEBG, ICEMS

C      WHERE TO PUT THE HEADING

              XVERS = FLOAT(VERSION)/100.
              CALL WINDOW (0, 0, 23, 79, ICBG, ICHDR)

C      HEADING, TIME, DATE AND VERSION STAMP

              LBUF = ' '
              WRITE(LBUF, 1) XVERS
           1  FORMAT('TWINDOW v',F4.1)
           2  FORMAT(128A1)
              LBUF (73:80) = DATOUT(1:8)

              CALL MESSNS (0, 0, LBUF, 80, ICMBG, ICHDR)
              RETURN
              END

              SUBROUTINE MESSNS (ROW, COL, PHRASE, Z, BACK, FORE)

C      THIS SUBROUTINE IS WRITTEN TO SIMPLIFY THE WRITING OF A LITERAL
C      STRING OF CHARACTERS.  USING THIS ROUTINE CUTS DOWN ON THE
C      NUMBER OF WRITE/FORMAT STATEMENTS THAT MUST BE USED.

              INTEGER Z, ROW, COL, BACK, FORE
              CHARACTER PHRASE*(*)

              CALL WINDOW (ROW, COL, ROW, COL+Z-1, BACK, FORE)
              CALL CHRMOV (ROW, COL)
              CALL MESSNR (PHRASE, Z)
C
              RETURN
              END

              SUBROUTINE NPUTKB (SPECIAL, SPKEY, MESSAGE, ML)

C      READ IN A STRING - THIS CALL WILL ACCEPT ALPHANUMERIC INPUT

              CHARACTER MESSAGE*(*)
              INTEGER SPKEY
              INTEGER*2 SP
              LOGICAL SPECIAL

              COMMON /TW/ ICHDR,ICSUB,ICTXT,ICPRO,ICMSG,ICHLP,ICBG,ICMBG,
```

```
      .  ICHBG, ICEBG, ICEMS

         CALL WINDOW (23, 0, 24, 79, ICBG, ICTXT)
         CALL CHRMOV (23, 0)
         CALL MESSNR
      . ('Use the cursor keys to move around the screen or <esc> to quit'
      . , 62)
         CALL CHRMOV (24, 0)
         CALL MESSNR (MESSAGE, ML)

4        CALL READKB (SP)

C        WAIT FOR A <CR> OR OTHER SPECIAL CHARACTER

         IF(SP.EQ.0) GO TO 4
         SPECIAL = .TRUE.
         SPKEY = SP
         CALL CURSOF
         RETURN
         END

         SUBROUTINE READKB (SP)

         INTEGER*2 CH, SP, HIT, TABLE(31)

         DATA TABLE /59,60,61,62,63,64,65,66,67,68,27,71,
      *            72,73,75,77,79,80,81,82,83,84,85,86,87,88,89,90,
      *            91,92,93/

C        THE FOLLOWING EXTENDED KEY CODES ARE RECOGNIZED AND RETURNED:
C              KEY        DEC. CODE       TABLE #
C              F1-F10     (59-68)          1-10
C              ESCAPE     (27)             11
C              HOME       (71)             12
C              UP  ARROW  (72)             13
C              PAGE UP    (73)             14
C              LFT ARROW  (75)             15
C              RT  ARROW  (77)             16
C              END        (79)             17
C              DWN ARROW  (80)             18
C              PAGE DWN   (81)             19
C              INSERT     (82)             20
C              DELETE     (83)             21
C              SHIFT Fn   (84-93)         22-31

         SP = 0
         CALL GRABKY (CH, HIT)
         IF (HIT.NE.0) THEN
            IF (HIT.GT.1.OR.CH.EQ.27) THEN
C
C           SPECIAL CODE
C
            J = 0
10          J = J+1
            IF (CH.EQ.TABLE(J)) THEN
C              SPECIAL CODE RECOGNIZED
               SP = J
            ELSE IF (J.LT.22) THEN
               GO TO 10
            ENDIF
         ELSE
            IF (CH.EQ.13) THEN
C
C           END OF STRING
```

```
         ENDIF
      ENDIF
   ENDIF
   RETURN
   END

   SUBROUTINE SWINDOW(TR, LC, BR, RC)

   LOGICAL SAVEINPT, SAVETHRM, ADVFEA
   INTEGER GTRAIL, FASTFOR, VERSION, LOGERR, SAVECL(11), SAVEUN(7)
   COMMON /FIN006/GTRAIL,FASTFOR,SAVEINPT,SAVETHRM,LOGERR,VERSION,
  . ADVFEA,SAVECL,SAVEUN

C  SAVE A WINDOW

   INTEGER TR, BR, LC, RC, ROW
   INTEGER ACTIVE, OFFSET(10), R(10), C(10), LEN(10), WID(10)
   INTEGER*2 BUFFER(6000)

   DATA ACTIVE/0/

   IF (ACTIVE.GE.10) STOP 'TOO MANY WINDOWS ACTIVE'

C  GENERATE THE STATS FOR THIS BUFFER

   ACTIVE = ACTIVE+1
   IF (ACTIVE.EQ.1) THEN
      OFFSET(ACTIVE) = 1
   END IF
   R(ACTIVE) = TR
   C(ACTIVE) = LC
   WID(ACTIVE) = RC-LC+1
   LEN(ACTIVE) =  BR-TR+1

C  STORE THE TEXT UNDER THE WINDOW TO BUFFER

   I = OFFSET(ACTIVE)
   ISTART = I
11 FORMAT('SWINDOW - SCREEN SPACE -> ',2I10)

   DO 10 ROW=TR,BR
      CALL VSTOW(BUFFER(I),ROW,LC,WID(ACTIVE))
      I = I+WID(ACTIVE)
      IF (I.GT.6000) STOP 'BUFFER SPACE EXCEEDED'
10 CONTINUE
   IF (ACTIVE.LT.10) THEN
      OFFSET(ACTIVE+1) = I+1
   END IF
   RETURN

   ENTRY RWINDOW

C  RESTORE THE PREVIOUS WINDOW

   IF (ACTIVE.LT.1) RETURN

   I = OFFSET(ACTIVE)
   DO 20 ROW = R(ACTIVE), R(ACTIVE)+LEN(ACTIVE)-1
      CALL VSHOW(BUFFER(I),ROW,C(ACTIVE),WID(ACTIVE))
      I = I+WID(ACTIVE)
20 CONTINUE
   ACTIVE = ACTIVE-1
   RETURN
   END
```

References

[1]   Jones,W.W. and Peacock,R.D., FAST Technical Reference Guide for Version 18, National Institute of Standards and Technology Technical Note 1262 (1990).

[2]   Jones,W.W. and Forney,G.P., A Programmer's Reference Manual for CFAST, the Unified Model of Fire Growth and Smoke Transport, to be published as a NISTIR.

[3]   Forney,G.P. and Cooper,L.Y., The Consolidated Compartment Fire Model (CCFM) Computer Application CCFM.VENTS - Part II: Software Reference Guide, National Institute of Standards and Technology Internal Report 90-4343 (1990).

4. TITLE AND SUBTITLE

Software Development Tools

5. AUTHOR(S)

Glenn P. Forney and Walter W. Jones

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This paper discusses the use of software tools to air in the development of models produced by the Center for Fire Research (CFR). There are two types of tools described in this paper. The first type consists of executable programs that characterize the subroutine and data structures of FORTRAN programs. A second class of software tools are subroutines that support various utility functions required by CFR models. We will discuss how each of these tools are used and why their development was necessary. In addition, we will indicate how these tools might be improved.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

compartment fires; fire growth; mathematical models; numerical models; software development